A new and significant technique for adapting computer system capacity to the workload

by Paul Abrahams

COMPILER PESSIMIZATION

The rapid increase in the capacity and operating speeds of computers is well known to all workers in the field. Until the recent business downturn dampened the demand for computing, this was widely heralded as "progress." Unfortunately, this progress has left the industry in real danger of finding itself with a large excess in computing capacity at the very time when computing needs are diminishing and management is issuing increasingly sharp memos on the need for frugality.

It is the contention of this paper that there are techniques available for absorbing this excess computer capacity and saving the edp department from the economy axe. There seems to be little doubt that these techniques will become increasingly significant in the '70s and should be mastered by any serious student of programming. Simply put, the techniques involve the incorporation of "make-work" into the object code produced by a compiler. Since most computer programming is now being done in higher-level languages, it is clear that the compilers for these languages are an obvious point of departure. Hence, compiler pessimization.

For make-work to be truly effective, it must have two characteristics:

1. It must be useless.
2. It must not be obviously useless.

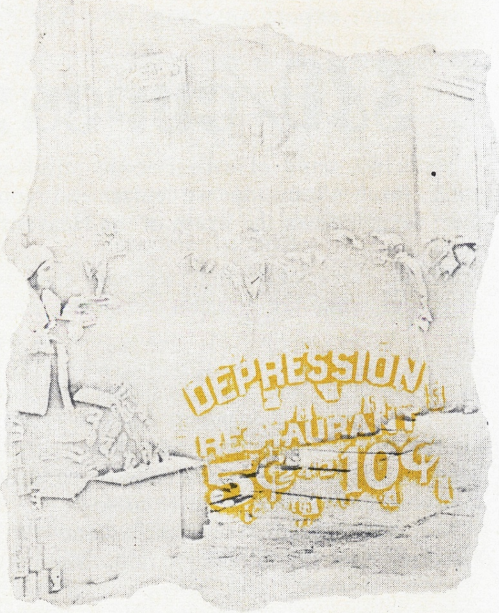
Thus, expanding computer usage by finding new applications or extending old ones is not satisfactory, for the results may inadvertently turn out to be useful. Nor will merely padding a compiled program with NOPs do; it is far too obvious that no operation can be accomplished by a "no operation" instruction. More subtle measures are called for.

We propose that the OPT parameter associated with the control languages for some compilers be expanded to permit negative values. Thus, OPT=0 would request that a program be neither optimized nor pessimized; OPT=-1 would call for primitive pessimization, and OPT=-2 would call for full pessimization, with the inefficiency of the compiled code being limited only by the inventiveness of the pessimizer. Existing programs, as well as new ones, could be pessimized through recompilation; thus the benefits of pessimization could be extended throughout a library of programs.

An interesting application of pessimization occurs when a computer is newly installed. With adequate pessimization, a small group of jobs can easily be made to strain the capacity of the machine virtually on the first day of operation. Thus the installation manager has a ready argument that an even larger machine is needed; or, alternatively, he can decrease the pessimization as new projects are attempted on the machine, thus keeping the load constant and the users satisfied.

To some extent, the possibilities for pessimization are language-dependent. The richer the language, the richer the potential for pessimization. Thus PL/I, for instance, with its elaborate block structure, proliferation of data types, and multitasking facilities, provides a vast new array of possibilities for the astute pessimizer. Indeed, there are those who maintain that PL/I compilations are naturally pessimized, without special effort in that direction.

One point must be made clear: the aim of pessimization is the creation of inefficiency and not the introduction of bugs. Thus, a pessimal program, like an optimal one, must produce the same results as the original one. Some specialists in the field would disagree with this approach, arguing that the introduction of bugs creates a high level of demand for programmers' services, starting with those of the application programmer, who must detect the bugs introduced through pessimization, and ending with the pessimizer himself, who must remove the offending code. While we do not dispute this conclusion, we maintain that our primary goal is to keep computers busy, not people; and production



runs are far more effective towards this end than are debugging runs. The advocates of the "introduce bugs" approach, however, can take comfort in the fact that pessimization is likely to introduce bugs at least inadvertently.

We now turn to the consideration of some specific pessimization techniques. We classify these according to whether they primarily affect central processor computation, input-output, or the use of storage.

The following are among the techniques known for the pessimization of central processor computation:

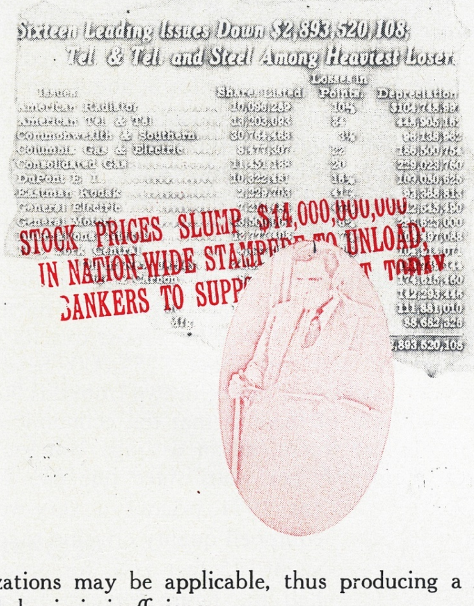
1. **Expansion of loops.** Code appearing outside of a loop may be moved into the loop, thus causing it to be executed many times instead of just once. For example, the FORTRAN code sequence:

```
      RHO = SIN(THETA)
      DO  10  K = 1,15
10    SIGMA = SIGMA + SQRT(RHO + G(K) )
```

may be altered to:

```
      DO  10  K = 1,15
      RHO = SIN(THETA)
10    SIGMA = SIGMA + SQRT(RHO + G(K) )
```

2. **Multiplication by constants.** Multiplication by integer constants may be reduced to repeated addition. Similarly, for cases where extreme pessimization is required, addition may be reduced to repeated incrementation by unity. Note that under some circumstances both the multiplication and addition pessimizations may be applicable, thus producing a cascaded gain in inefficiency.


Sixteen Leading Issues Down $2,893,520,108;
Tel & Tel and Steel Among Heaviest Losers
STOCK PRICES SLUMP $14,000,000,000
IN NATION-WIDE STAMPEDE TO UNLOAD;
BANKERS TO SUPPORT MARKET TODAY

3. **Injection of common subexpressions.** The FORTRAN sequence

```
   RHO = SIN(THETA)
   SIGMA1 = RHO + RHO**2
   SIGMA2 = RHO/3
```

may be converted to

```
   SIGMA1 = SIN(THETA + SIN(THETA)**2
   SIGMA2 = SIN(THETA)/3
```

thus causing the SIN to be computed three times instead of once. Better yet, the elimination of a statement creates the appearance of a **gain** in efficiency. Injection of common subexpressions also implies that subscript multiplication should be carried out for each array reference without regard to other array references or the known characteristics of the indexing variables.

4. **Register allocation.** Results should be moved back and forth from storage as often as possible, and fast registers should be used only when necessary to achieve correct results.

5. **Postponement of binding.** Expressions whose value is known at compile time can nevertheless be recomputed at run time. In the case of a block structure language, various expressions relating to the sizes of aggregates may become known at compile time, at the time of entrance to a block, or at the time of computing an expression. All such computations and the associated storage allocations should be made at expression time if possible; otherwise they should be made at block time. Under no circumstances should they be made at compile time.

Some of the input-output pessimization techniques may require modification of programs other than the compiler itself. The following are some known techniques:

1. **Page control.** In a paging environment, one should adopt the MRU (most recently used) algorithm. Under this algorithm, when a page must be expelled from core, the most recently used page is the one expelled. If the operating system permits compiled programs to expel pages, then for maximal pessimization, pages can be expelled after each reference.

2. **Choice of record size.** Record sizes should be selected so as to maximize short records, unfilled allocations, and interrecord gaps.

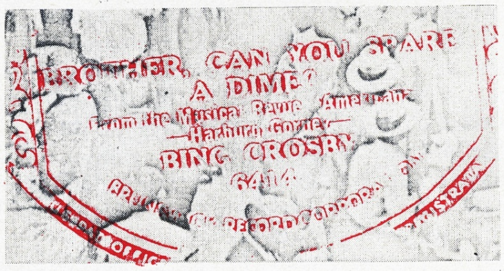
3. **Buffer management.** Input buffers should not be read into until they are entirely empty; similarly, output buffers should not be written out until they are entirely full.

The object in storage pessimization is to maximize the core requirements of the compiled program. Here are some methods:

1. **Allocation of temporary storage.** Temporary storage locations required for intermediate results should be assigned to separate locations, rather than being overlapped. Allocations, if done dynamically at run time, should be done in as small blocks as possible.

2. **Choice of buffer size.** Large buffers, when managed according to the principle suggested above, can consume arbitrary amounts of space and often can force an overlay structure to be used.

Acknowledgment. I am grateful to the marketing representatives of several large computer manufacturers, without whose encouragement this article would


BROTHER, CAN YOU SPARE A DIME?
From the Musical Revue "Americana"
—Harburg-Gorney—
BING CROSBY
6414

never have been written. Their enthusiastic response to the pessimization concept was a constant source of inspiration to me. At their request, however, they and the companies that they represent must remain nameless. ■